BITS Pilani

ML SYSTEM OPTIMIZATION

Session 3

M1: Foundations

Modern Hardware Architectures

innovate achieve lead

BITS Pilani

1. *Multi-core CPUs*
2. *GPGPUs*
3. *GPU/CPU Clusters*
4. *Resource Management*

Multi-core CPUs

- A **multi-core CPU** is a processor that contains two or more independent processing units called **cores** on a single chip.
- Each core can execute instructions independently, allowing multiple tasks or multiple parts of the same program to run simultaneously.
- Unlike traditional single-core processors, multicore processors improve performance through **parallel execution** rather than simply increasing clock speed.
- Modern computers, smartphones, servers, and AI systems rely heavily on multicore processors because many applications require concurrent execution of multiple tasks.

Why Multi-core CPUs?

- Increasing processor clock frequency was once the primary method for improving CPU performance. However, higher frequencies resulted in:
  - Increased power consumption
  - Greater heat generation
  - Reduced energy efficiency
  - Physical limitations in transistor switching
- Instead of making one core faster, manufacturers began placing multiple processing cores on a single processor chip.

This approach provides:

- Better multitasking
- Higher throughput
- Improved responsiveness
- Better energy efficiency

Work Integrated Learning Programmes

Types of Multi-core CPUs

- Multi-core processors are classified based on the number of processing cores integrated into a single CPU chip.
- As the number of cores increases, the processor can execute more tasks simultaneously, resulting in improved multitasking, higher throughput, and better performance for parallel applications.

Processor Type	Number of Cores	Typical Applications
Dual-core	2	Basic computing, web browsing, office applications, and light multitasking.
Quad-core	4	General-purpose computing, programming, multimedia, and moderate gaming.
Hexa-core	6	Content creation, software development, gaming, and data analysis.
Octa-core	8	High-performance gaming, video editing, virtualization, and scientific computing.
Many-core Processor	16, 32, 64, or more	Artificial Intelligence (AI), Machine Learning (ML), High-Performance Computing (HPC), cloud servers, large-scale simulations, and parallel scientific applications.

Multi-core CPUs

A **multi-core CPU** integrates multiple independent processing cores on a single chip.

Each Core Contains:

- Private **registers**
- **ALU (Arithmetic Logic Unit)**
- Often private **L1 cache**
- Shared **L2/L3 cache**
- Shared **main memory (RAM)**

Parallel Model → Shared Memory Model

All cores access the same global address space:

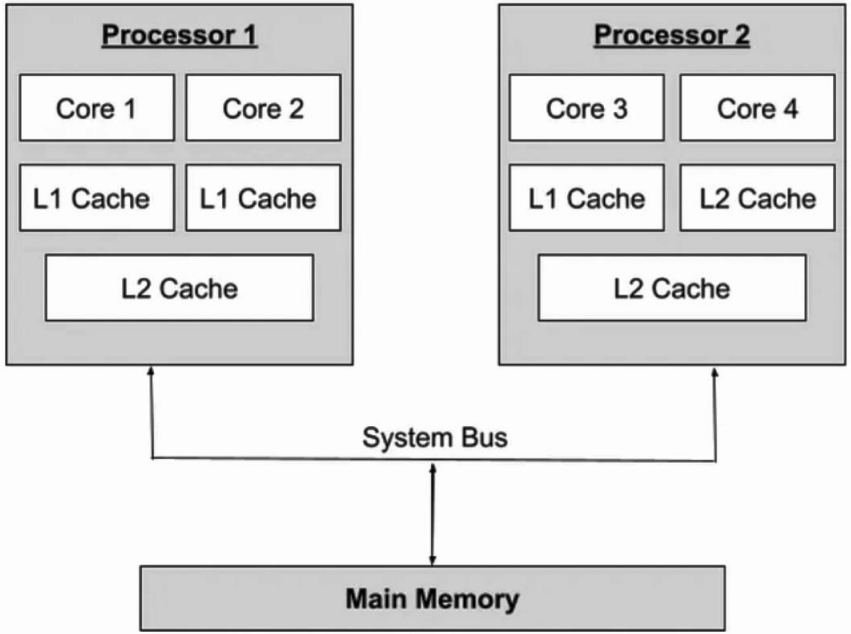
$$M_{global} \rightarrow \{Core_1, Core_2, \ldots, Core_N\}$$

This enables:

- Thread-based programming (e.g., OpenMP, Pthreads)
- Fast inter-thread communication (via shared variables)

Work Integrated Learning Programmes

Multi-core CPUs



CPU Cores

Each CPU core typically contains:

- **Arithmetic Logic Unit (ALU)** – Performs arithmetic and logical operations.
- **Control Unit (CU)** – Directs and coordinates instruction execution.
- **Registers** – High-speed storage locations for temporary data and instructions.
- **Instruction Pipeline** – Enables multiple instruction stages to execute concurrently, increasing throughput.
- **L1 Cache** – Small, high-speed private cache for frequently accessed instructions and data.
- **Branch Prediction Unit** – Predicts the outcome of branch instructions to minimize pipeline stalls.
- **Instruction Decoder** – Converts machine instructions into internal operations.
- **Load/Store Unit** – Handles data transfer between registers and memory.
- **Floating Point Unit (FPU)** – Executes floating-point arithmetic operations.

CPU Cores Characteristics

- **Independent Core Execution:** Each CPU core executes its own instruction stream independently, enabling concurrent processing of different tasks.
- **True Parallel Processing:** Multiple cores can execute different programs, processes, or threads simultaneously, significantly improving overall system performance and throughput.
- **Simultaneous Multithreading (SMT):** Many modern processors support **SMT**, allowing a single physical core to execute multiple hardware threads concurrently, improving resource utilization and execution efficiency.

Work Integrated Learning Programmes

Cache Memory

Cache memory is a small, high-speed memory that stores frequently accessed instructions and data, reducing access time compared to main memory.

L1 Cache (Level 1 Cache)

- **Integrated Within the CPU Core:** Located directly inside each processor core.
- **Private Cache:** Each core has its own dedicated L1 cache, which is not shared with other cores.
- **Fastest Cache Level:** Provides the lowest latency and highest access speed among all cache levels.
- **Small Capacity:** Typically ranges from **32 KB to 128 KB per core**.
- **Split Cache Architecture:** Usually divided into two separate caches:
  - **L1 Instruction Cache (L1I):** Stores frequently executed program instructions.
  - **L1 Data Cache (L1D):** Stores frequently accessed data required by the executing instructions.
- **Purpose:** Minimizes the time required for the CPU to fetch instructions and data, enabling faster program execution.

Cache Memory

innovate achieve lead

L2 Cache (Level 2 Cache)

- **Larger Capacity:** Provides more storage than the L1 cache, typically ranging from **256 KB to several MB** per core or cluster.
- **Moderate Speed:** Slower than L1 cache but significantly faster than accessing main memory (RAM).
- **Core Organization:** May be **dedicated to a single CPU core** or **shared among a small group of cores**, depending on the processor architecture.
- **Secondary Cache Level:** Stores frequently accessed instructions and data that are not available in the L1 cache.
- **Reduces Memory Accesses:** Acts as an intermediate buffer between the L1 cache and L3 cache (or RAM), reducing the need to access slower memory.
- **Purpose:** Improves processor performance by supplying data to the L1 cache more quickly than fetching it directly from main memory.

Cache Memory

L3 Cache (Level 3 Cache / Shared Cache)

- **Shared Cache:** Accessible by all processor cores, enabling efficient data sharing across the CPU.
- **Largest On-Chip Cache:** Provides a much larger storage capacity than L2 cache, typically ranging from **several MB to tens of MB**.
- **Higher Latency:** Slower than both L1 and L2 caches, but still significantly faster than accessing main memory (RAM).
- **Last-Level Cache (LLC):** Acts as the final cache layer before data is fetched from RAM.
- **Reduces RAM Accesses:** Stores frequently used instructions and data that are not present in L1 or L2, minimizing costly accesses to main memory.
- **Facilitates Inter-Core Communication:** Allows processor cores to efficiently share data, reducing duplication and improving the performance of multi-core applications.
- **Purpose:** Enhances overall system performance by decreasing memory latency and improving data availability for all CPU cores.

Cache Memory

Last-Level Cache (LLC)

The **Last-Level Cache (LLC)** is the final cache level that the processor checks before accessing the slower main memory (RAM). It reduces memory access latency and improves overall system performance.

Characteristics

- **Final Cache Level:** The last cache accessed before retrieving data from RAM.
- **Architecture-Dependent:** The LLC varies based on the processor design.
- **Typically L3 Cache:** In most modern processors, the **L3 cache** serves as the LLC.
- **Optional L4 Cache:** Some processors include an **L4 cache**, which becomes the LLC.
- **Shared Cache:** Typically shared among multiple CPU cores for efficient data sharing.
- **Reduces RAM Accesses:** Stores frequently used data and instructions not found in L1 or L2 caches.
- **Improves Performance:** Increases cache hit rates and reduces memory access time.

Memory Controller

The **Memory Controller** manages communication between the CPU and the system's main memory (RAM), ensuring efficient and reliable data transfers.

Functions

- **Handles Memory Requests:** Receives memory requests from CPU cores.
- **Address Management:** Translates and manages memory addresses.
- **Memory Access Scheduling:** Schedules memory read and write operations.
- **DRAM Control:** Manages DRAM timing and synchronization.
- **Bandwidth Optimization:** Optimizes memory bandwidth for efficient data transfer.

Modern Memory Controllers

Modern processors integrate the memory controller directly onto the CPU chip, providing:

- **Lower Memory Latency:** Faster access to main memory.
- **Higher Memory Bandwidth:** Increased data transfer rates.
- **Improved System Performance:** More efficient CPU–memory communication.

Work Integrated Learning Programmes

Interconnection Network

The **Interconnection Network** provides the communication infrastructure that connects CPU cores, shared caches, memory controllers, and other processor components.

Functions:

- Transferring data between cores.
- Connecting cores to shared caches.
- Routing memory requests.
- Maintaining cache coherence.
- Supporting synchronization between cores.

Work Integrated Learning Programmes

Interconnection Network

Ring Bus

- Cores are connected in a circular ring.
- Data travels around the ring until it reaches its destination.
- Simple and efficient for processors with a moderate number of cores.

Mesh Network

- Cores are arranged in a two-dimensional grid.
- Data is routed through neighboring cores.
- Highly scalable for processors with many cores.

Crossbar Interconnect

- Provides direct communication paths between all connected components.
- Offers very low latency.
- Hardware complexity increases rapidly as the number of cores grows.

Network-on-Chip (NoC)

- Treats the processor like a miniature computer network.
- Uses routers and communication links to transfer data packets.
- Highly scalable and widely used in many-core processors.

Work Integrated Learning Programmes

Main Memory (RAM)

Main Memory (RAM) stores the programs and data required by all CPU cores during program execution. It provides the primary working memory for the processor.

Characteristics

- **Shared Memory:** Accessible by all processor cores.
- **Large Capacity:** Provides significantly more storage than cache memory.
- **Higher Latency:** Slower than L1, L2, and L3 caches.
- **Accessed on Cache Miss:** Data is retrieved from RAM only when it is unavailable in the processor caches.
- **Supports Concurrent Access:** Multiple CPU cores can access the same memory simultaneously.
- **Maintains Data Consistency:** Memory consistency and cache coherence mechanisms ensure that all cores observe a consistent and up-to-date view of shared data.

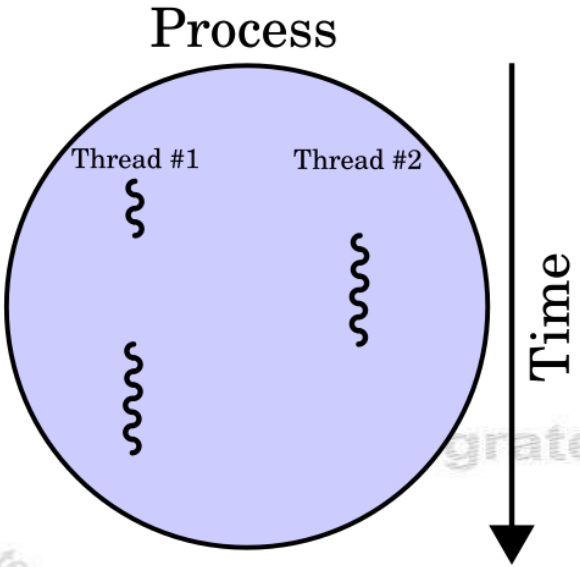
Thread-based Programming

- A **thread** is the smallest unit of execution that can be scheduled by the operating system. A process may contain one or more threads, allowing multiple tasks to execute concurrently within the same application.
- Unlike separate processes, threads belonging to the same process **share the same address space and system resources**, enabling efficient communication and lower execution overhead.
- In multi-core processors, multiple threads can execute simultaneously on different CPU cores, significantly improving performance for parallel applications.

Process vs Thread

Feature	Process	Thread
Definition	An independent executing program	A lightweight execution unit within a process
Memory	Own address space	Shares the process address space
Resource Sharing	Does not share memory with other processes	Shares memory and resources with other threads
Creation Cost	High	Low
Context Switching	Expensive	Faster and less expensive
Communication	Uses Inter-Process Communication (IPC)	Direct communication through shared memory
Failure Impact	Process failure usually does not affect others	One thread failure may affect the entire process
Typical Usage	Independent applications	Parallel tasks within an application

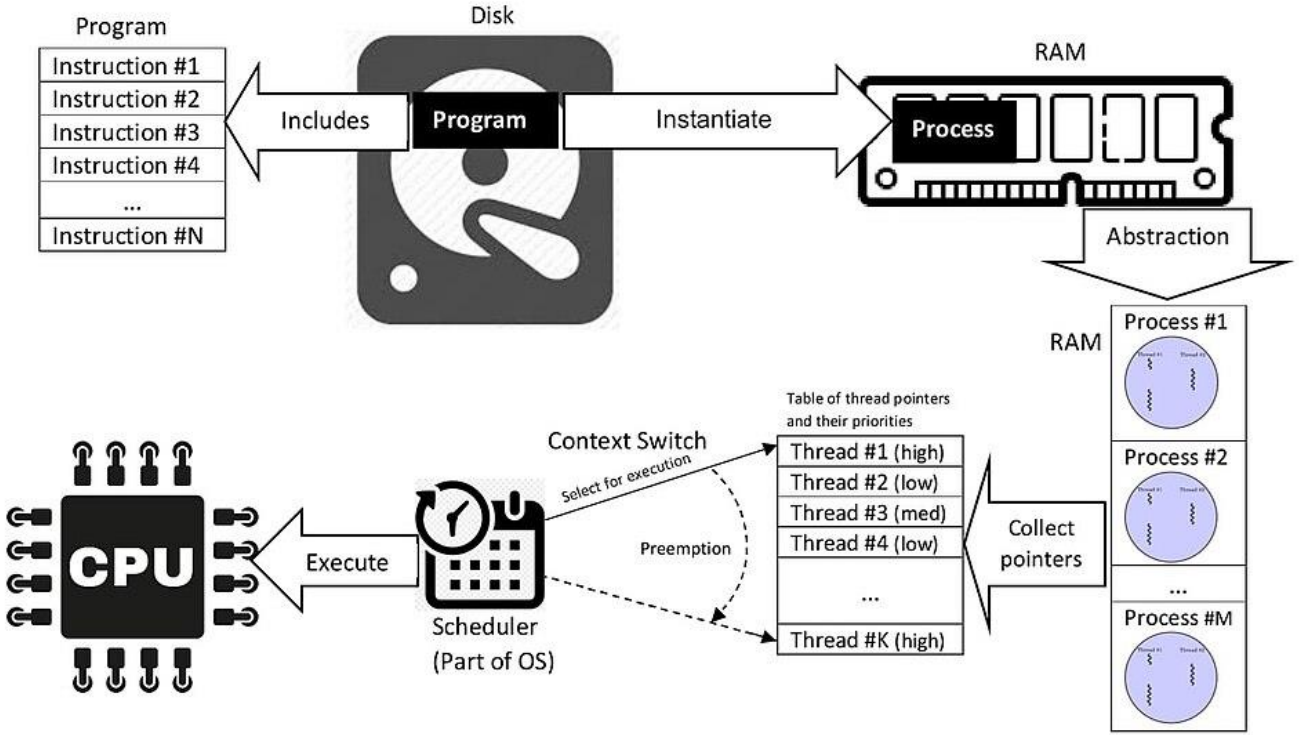
Process vs Thread



A process with two threads of execution, running on one processor

- Multiple **threads run concurrently** within the same process.
- Threads **share the same memory space** of the process.
- Shared memory allows **faster communication and data exchange** between threads.
- **Thread #1 and Thread #2** act as independent execution paths working in parallel.
- Multithreading is commonly used in **parallel computing, operating systems, and high-performance applications**.

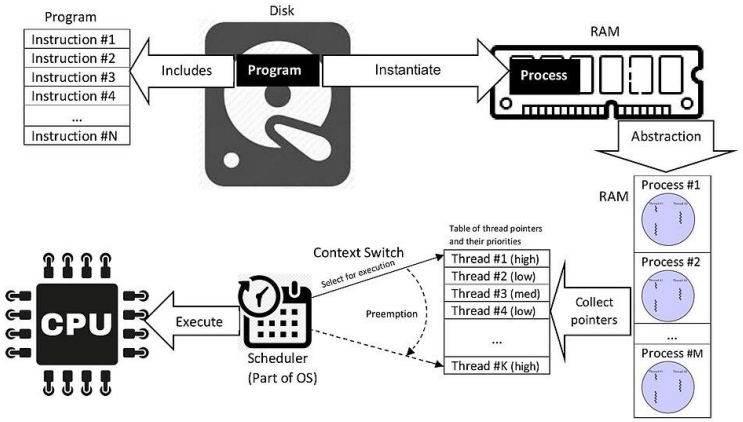
Process vs Thread



Process vs Thread

Execution Flow

- **Program:** Source code is compiled into an executable file.
- **Storage:** Executable is stored on **Disk (SSD/HDD)**.
- **Execution:** User launches the program.
- **Loading:** OS loads the program from **Disk → RAM**.
- **Process:** Program becomes a **running process** with its own memory and resources.
- **Threads:** Process creates one or more **threads** for execution.
- **Ready Queue:** Runnable threads are placed in the scheduler's queue.
- **Scheduling:** OS scheduler selects the next thread based on priority/scheduling policy.
- **CPU Execution:** Selected thread executes using the **Fetch → Decode → Execute** cycle.
- **Preemption:** CPU switches when the time slice expires or a higher-priority thread arrives.
- **Context Switch:** Current thread state is saved; next thread state is restored.
- **Completion:** Process continues until execution finishes or is terminated.



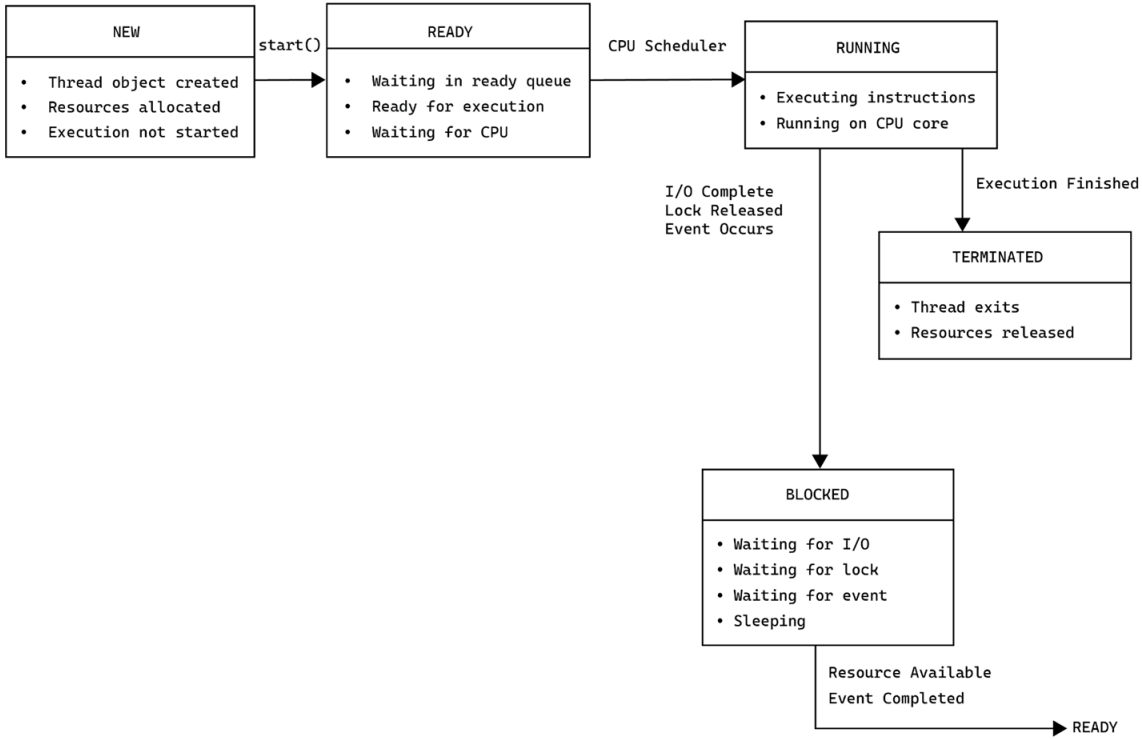
Thread Scheduling in Multi-core CPUs

Thread scheduling is the process by which the operating system assigns runnable threads to available CPU cores for execution, ensuring efficient utilization of processor resources.

Scheduling Behavior

- **Single-Core Processor:** Only one thread executes at a time. Multiple threads are executed using **time slicing**, creating the illusion of concurrent execution.
- **Multi-Core Processor:** Multiple threads can execute simultaneously on different CPU cores, enabling **true parallel execution**.
- **Oversubscription:** When the number of runnable threads exceeds the number of available CPU cores, the operating system uses **context switching** to alternate thread execution across the cores.
- **Efficient Core Utilization:** The scheduler balances the workload across CPU cores to improve performance and maximize processor utilization.

Thread Lifecyle



Instruction Pipelining

- **Instruction pipelining** is a CPU design technique that improves processor performance by **executing multiple instructions simultaneously at different stages of execution**.
- Instead of waiting for one instruction to complete before starting the next, the processor divides instruction execution into several stages, allowing different instructions to occupy different stages at the same time.
- Pipelining exploits **Instruction-Level Parallelism (ILP)**, enabling the CPU to achieve higher throughput without duplicating the entire processor.

Work Integrated Learning Programmes

Why is Pipelining Needed?

In a non-pipelined processor, an instruction must complete all execution steps before the next instruction begins.

Without Pipelining

Instruction 1 → Complete
Instruction 2 → Complete
Instruction 3 → Complete

Only one instruction is processed at any instant.

This leads to

- Poor hardware utilization
- Idle functional units
- Lower instruction throughput
- Longer program execution time

Instruction pipelining overcomes these limitations by allowing different hardware units to work simultaneously.

Instruction Execution Stages

An instruction passes through several sequential stages.

Typical execution stages are

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

Instead of completing one instruction before starting the next, the CPU overlaps these stages.

Work Integrated Learning Programmes

Instruction Execution Stages

Instruction	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
I_1	IF	ID	EX	MEM	WB				
I_2		IF	ID	EX	MEM	WB			
I_3			IF	ID	EX	MEM	WB		
I_4				IF	ID	EX	MEM	WB	
I_5					IF	ID	EX	MEM	WB

- **IF** – Instruction Fetch
- **ID** – Instruction Decode
- **EX** – Execute
- **MEM** – Memory Access
- **WB** – Write Back

Work Integrated Learning Programmes

Instruction Execution Stages

Clock Cycle	Pipeline Activity	Status
Cycle 1	$I_1 \rightarrow$ IF	The first instruction (I_1) enters the pipeline and is fetched from instruction memory. Only one stage is active.
Cycle 2	$I_1 \rightarrow$ ID $I_2 \rightarrow$ IF	I_1 is decoded while I_2 is fetched. Two instructions are processed simultaneously.
Cycle 3	$I_1 \rightarrow$ EX $I_2 \rightarrow$ ID $I_3 \rightarrow$ IF	I_1 executes, I_2 is decoded, and I_3 is fetched. Three pipeline stages are active concurrently.
Cycle 4	$I_1 \rightarrow$ MEM $I_2 \rightarrow$ EX $I_3 \rightarrow$ ID $I_4 \rightarrow$ IF	Four instructions are in different stages simultaneously. The pipeline is almost full.
Cycle 5	$I_1 \rightarrow$ WB $I_2 \rightarrow$ MEM $I_3 \rightarrow$ EX $I_4 \rightarrow$ ID $I_5 \rightarrow$ IF	The pipeline becomes fully occupied. All five stages are active, and I_1 completes execution.
Cycle 6	$I_2 \rightarrow$ WB $I_3 \rightarrow$ MEM $I_4 \rightarrow$ EX $I_5 \rightarrow$ ID	I_2 completes execution while the remaining instructions advance to the next stages. One instruction finishes in this cycle.
Cycle 7	$I_3 \rightarrow$ WB $I_4 \rightarrow$ MEM $I_5 \rightarrow$ EX	I_3 completes execution. The pipeline begins to drain since no new instructions enter.
Cycle 8	$I_4 \rightarrow$ WB $I_5 \rightarrow$ MEM	I_4 completes execution, while I_5 performs memory access. Only two instructions remain in the pipeline.
Cycle 9	$I_5 \rightarrow$ WB	I_5 completes execution. The pipeline becomes empty after all instructions have finished.

- **Cycles 1–4:** Pipeline **filling phase**.
- **Cycle 5:** Pipeline reaches **maximum utilization** (all five stages are occupied).
- **Cycles 6–9:** Pipeline **draining phase**.
- After the pipeline is filled, **one instruction completes in every clock cycle**, resulting in higher CPU throughput.

Clock Cycle Time

- A **clock cycle** is the basic unit of time used by a processor to perform operations. Each instruction execution is divided into a sequence of clock cycles, where the processor's internal components perform specific tasks synchronized with the clock signal.
- **Clock cycle time** is the time required for the processor clock to complete one full cycle (one clock period). It represents the duration between two consecutive clock pulses.
- The relationship between clock frequency and clock cycle time is:

$$T_{cycle} = \frac{1}{f}$$

Where:

- T_{cycle} = Clock cycle time (seconds)
- f = Clock frequency (Hz)

Clock Cycle Time

If a processor operates at:

$$f = 2\ \text{GHz} = 2 \times 10^9\ \text{Hz}$$

Then,

$$T_{cycle} = \frac{1}{2 \times 10^9}$$

$$T_{cycle} = 0.5 \times 10^{-9}\ \text{seconds}$$

$$T_{cycle} = 0.5\ \text{ns}$$

Therefore, a **2 GHz processor has a clock cycle time of 0.5 nanoseconds**.

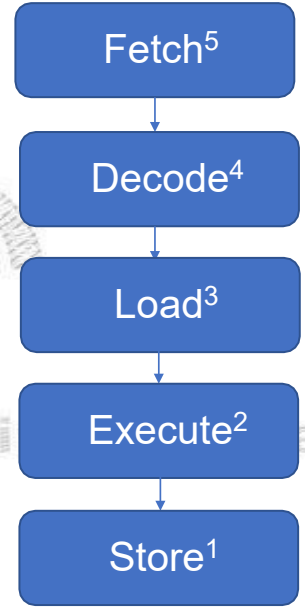
Clock cycle time determines how quickly a processor can execute individual stages of instruction processing. In a pipelined processor, each pipeline stage operates within one clock cycle, so reducing the clock cycle time increases the overall instruction throughput.

Multi-core CPUs - Instruction Pipeline

Given a sequence of instructions of the form:

- I1
- I2
- I3
- I4
- I5
- …

execution in a pipeline would appear thus ==>



If each stage takes 1 clock cycle, then throughput has increased:

- from 1 instruction per 5 cycles (sequential)
- to 1 instruction per 1 cycle (pipelined)

Q: What about Turn-around-Time aka response time?

Executing an Instruction

- Different stages of Instruction Execution:

Fetch Instruction	*Move the next instruction (tracked by PC) to a register*
Decode Instruction	*Identify Operator and (data) addresses*
Load (data)	*Move data into register (if needed)*
Execute	*Perform the operation*
Store (result)	*Move the resulting data to memory*

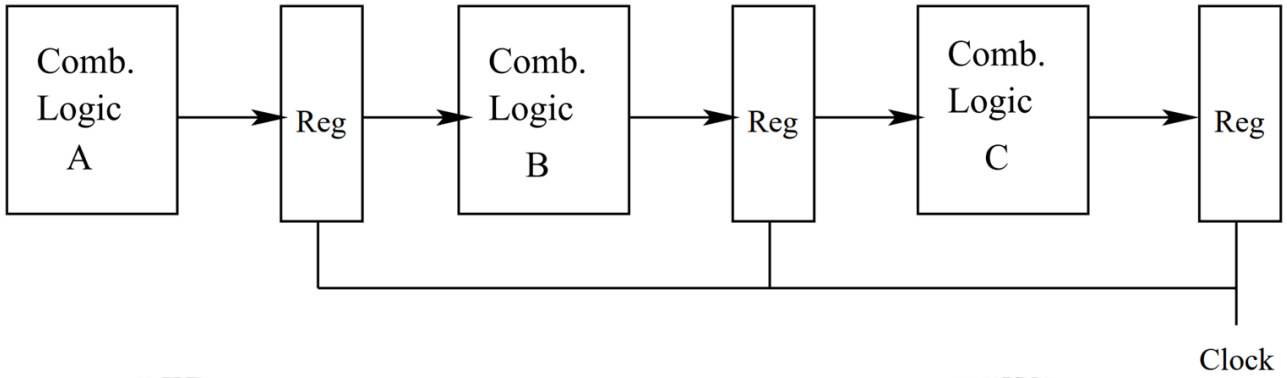
If separate circuitry is designed for each stage-
so that the stage take the same amount of time -
then a sequence of instructions can be executed in a pipeline

Pipeline Registers

Between every two stages, the processor contains **pipeline registers**.

These registers

- Store intermediate results
- Synchronize data between stages
- Separate combinational logic
- Enable independent operation of each stage



SIMD – Single Instruction, Multiple Data

Executes the **same instruction** across multiple data elements

Key Features:

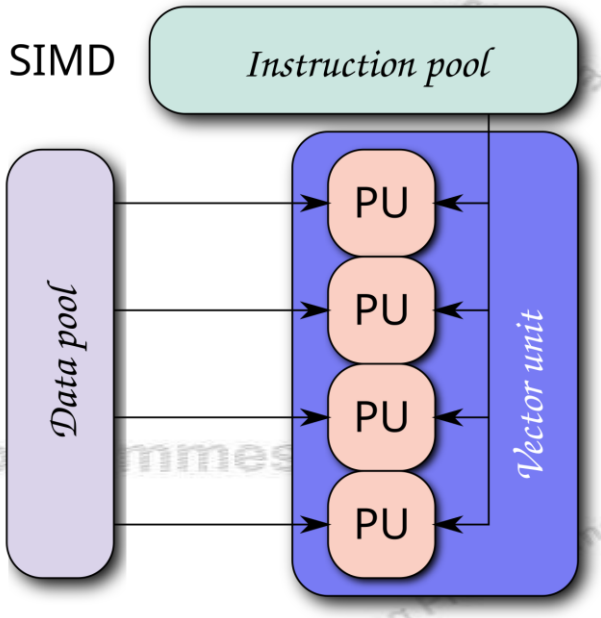
- Suitable for **data-parallel** applications
- Lockstep execution – all processing elements perform same operation simultaneously

Examples of Use:

- Vector and matrix operations
- Image filters, audio processing
- Scientific computation

Hardware Support:

- Intel AVX, ARM NEON, NVIDIA CUDA (warps/wavefronts)



Vector Addition

In SIMD, a single instruction performs the same mathematical operation on multiple data elements simultaneously.

Given two vectors:

$$A = [a_1 \quad a_2 \quad a_3 \quad a_4]$$

$$B = [b_1 \quad b_2 \quad b_3 \quad b_4]$$

A SIMD addition instruction performs:

$$C = A + B$$

where:

$$C = [c_1 \quad c_2 \quad c_3 \quad c_4]$$

and each element is calculated as:

$$c_i = a_i + b_i$$

Scalar vs SIMD Execution

Scalar processing:

$$4 \text{ additions} = (a_1 + b_1), (a_2 + b_2), (a_3 + b_3), (a_4 + b_4)$$

Requires:

$$4 \text{ instructions}$$

SIMD processing:

$$[A_1, A_2, A_3, A_4] + [B_1, B_2, B_3, B_4]$$

Requires:

$$1 \text{ SIMD instruction}$$

Scalar vs SIMD Execution

Speedup:

$$\text{Speedup} = \frac{\text{Number of scalar operations}}{\text{Number of SIMD operations}}$$

For a 4-element vector:

$$\text{Speedup} = \frac{4}{1} = 4\times$$

Thus, SIMD can theoretically provide a **4× performance improvement** for operations that can be fully parallelized.

MIMD – Multiple Instruction, Multiple Data

Each processor or core can execute **different instructions** on **different data**

Key Features:

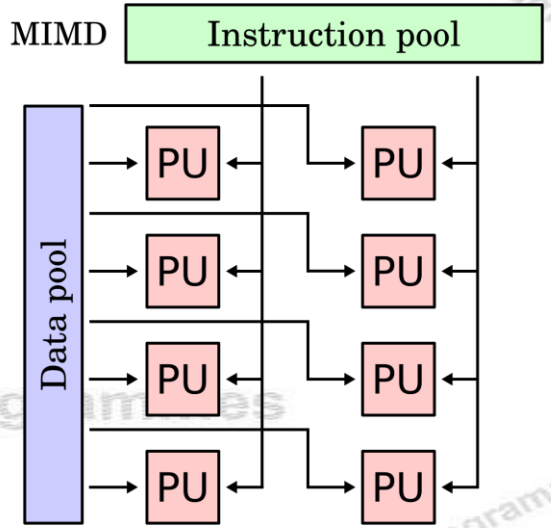
- Flexible and versatile execution model
- Ideal for task-parallel workloads

Examples of Use:

- Operating systems
- Web servers
- Multithreaded applications

Hardware Support:

- Multicore CPUs (Intel, AMD)
- Some heterogeneous compute platforms (e.g., ARM big.LITTLE)



MIMD – Multiple Instruction, Multiple Data

Assume there are **n processors**:

$$P_1, P_2, P_3, \ldots, P_n$$

Each processor executes its own instruction:

$$I_1, I_2, I_3, \ldots, I_n$$

on different data:

$$D_1, D_2, D_3, \ldots, D_n$$

The execution can be represented as:

$$P_i : I_i(D_i) \rightarrow R_i$$

where:

- P_i = Processor/Core i
- I_i = Instruction executed by processor i
- D_i = Input data for processor i
- R_i = Result produced by processor i

MIMD – Multiple Instruction, Multiple Data

innovate achieve lead

Example: Different Operations on Different Data

In a multi-core CPU, each core can execute a different instruction on independent data at the same time. This demonstrates **task-level parallelism**, where multiple cores perform separate computations concurrently.

Consider four CPU cores:

$$P_1 : R_1 = A_1 + B_1$$

$$P_2 : R_2 = A_2 \times B_2$$

$$P_3 : R_3 = A_3 - B_3$$

$$P_4 : R_4 = \frac{A_4}{B_4}$$

Execution flow:

$$\begin{array}{cccc} P_1 & P_2 & P_3 & P_4 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ (A_1, B_1) & (A_2, B_2) & (A_3, B_3) & (A_4, B_4) \\ + & \times & - & \div \\ \downarrow & \downarrow & \downarrow & \downarrow \\ R_1 & R_2 & R_3 & R_4 \end{array}$$

MIMD – Multiple Instruction, Multiple Data

$$
\begin{array}{cccc}
P_1 & P_2 & P_3 & P_4 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
(A_1, B_1) & (A_2, B_2) & (A_3, B_3) & (A_4, B_4) \\
+ & \times & - & \div \\
\downarrow & \downarrow & \downarrow & \downarrow \\
R_1 & R_2 & R_3 & R_4
\end{array}
$$

- **Core** P_1 performs addition using inputs A_1 and B_1, producing output R_1.
- **Core** P_2 performs multiplication using inputs A_2 and B_2, producing output R_2.
- **Core** P_3 performs subtraction using inputs A_3 and B_3, producing output R_3.
- **Core** P_4 performs division using inputs A_4 and B_4, producing output R_4.

Since each operation is independent, all four CPU cores execute these instructions **simultaneously**. A single-core processor would execute these operations one after another, while a multi-core processor completes them in parallel, improving execution speed and overall throughput.

MIMD – Multiple Instruction, Multiple Data

Matrix Representation

The independent operations performed by multiple processors can be represented using a result vector:

$$R = \begin{bmatrix} R_1(D_1) \\ R_2(D_2) \\ R_3(D_3) \\ \vdots \\ R_n(D_n) \end{bmatrix}$$

where each processor generates an independent output:

$$R_i = f_i(D_i)$$

Here:

- R_i represents the output produced by the i^{th} processor.
- D_i represents the input data assigned to the i^{th} processor.
- f_i represents the computation function executed by the i^{th} processor.

Each processor operates independently on its own data set, and the function f_i may differ for every processor. Therefore, multiple computations can be executed simultaneously, improving parallel processing efficiency.

SIMD vs. MIMD – Key Differences

Aspect	SIMD	MIMD
Instruction Stream	Single	Multiple
Data Stream	Multiple	Multiple
Synchronization	High (lockstep)	Low (independent threads)
Control Flow	Less flexible	Highly flexible
Hardware Examples	GPUs, Vector Units	Multicore CPUs, Clusters
Use Cases	Matrix ops, signal processing	Web servers, OS-level tasks
Efficiency	High for regular data structures	High for diverse workloads

Work Integrated Learning Programmes

SIMD vs. MIMD – Key Differences

Architecture	Mathematical Representation	Explanation
SIMD (Single Instruction Multiple Data)	$I(D_1, D_2, D_3, D_4) \rightarrow (R_1, R_2, R_3, R_4)$ $R_i = I(D_i)$	A **single instruction** I is applied to multiple data elements D_1, D_2, D_3, D_4 simultaneously, producing multiple outputs R_1, R_2, R_3, R_4.
MIMD (Multiple Instruction Multiple Data)	$I_1(D_1), I_2(D_2), I_3(D_3), I_4(D_4) \rightarrow (R_1, R_2, R_3, R_4)$ $R_i = I_i(D_i)$	**Multiple instructions** I_1, I_2, I_3, I_4 operate independently on different data elements D_1, D_2, D_3, D_4, producing separate outputs.

Work Integrated Learning Programmes

Performance Model

If a program has:

- Parallel fraction = P
- Number of cores = N

Then theoretical speedup:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

- Barriers(1−P) → Sequential part (cannot be parallelized)
- P / N → Parallel part divided across N cores

$$T_{parallel} = T_{compute} + T_{synchronization} + T_{memory\ contention}$$

$T_{compute}$: Actual useful computation time.

$T_{synchronization}$: Time spent:

- Waiting for other threads
- Barriers
- Locks / coordination

T_{memory} : Time lost due to:

- Shared memory access conflicts
- Cache coherence traffic
- Limited memory bandwidth

Memory Hierarchy Impact

CPU performance depends heavily on **cache behavior**.

Access times:

$$T_{L1} < T_{L2} < T_{L3} < T_{RAM}$$

Since RAM access is much slower, cache performance dominates CPU speed

Effective memory access time (EMAT):

$$EMAT = H \cdot T_{cache} + (1 - H) \cdot T_{memory}$$

Where:

- H = cache hit rate
- T_{cache} = cache access time
- T_{memory} = RAM access time

Higher H → lower average memory delay → better CPU performance.

Work Integrated Learning Programmes

Multi-core CPUs

Multi-core CPUs are effective for:

- Task-level parallelism
- Moderate data parallelism
- Branch-heavy computations
- General-purpose workloads

Multi-core performance is determined by:

$$Performance = Parallel\ Speedup - Synchronization + Memory\ Overhead$$

Even with many cores, scalability depends on:

- Parallel fraction P
- Synchronization cost
- Memory behavior
- Cache efficiency

GPGPUs

A GPGPU is designed for **massive parallel throughput computing**.

It contains:

- Thousands of lightweight cores
- Organized into **Streaming Multiprocessors (SMs)**
- Each SM executes many threads concurrently

CPU	GPU
Few powerful cores	Thousands of simpler cores
Optimized for low latency	Optimized for high throughput
Complex control logic	Minimal control logic
Large cache	Smaller cache, higher bandwidth

Execution Model – SIMT

SIMT - Single Instruction, Multiple Threads

One instruction is executed by many threads simultaneously.
Example:

$$C[i]= A[i] + B[i] \; C[i]$$

If:

- n elements
- T threads

Each thread computes one index independently. Since operations are independent, they execute in parallel.
This model is ideal when:

- Same operation
- Different data elements

Memory Hierarchy in Multicore & GPGPU

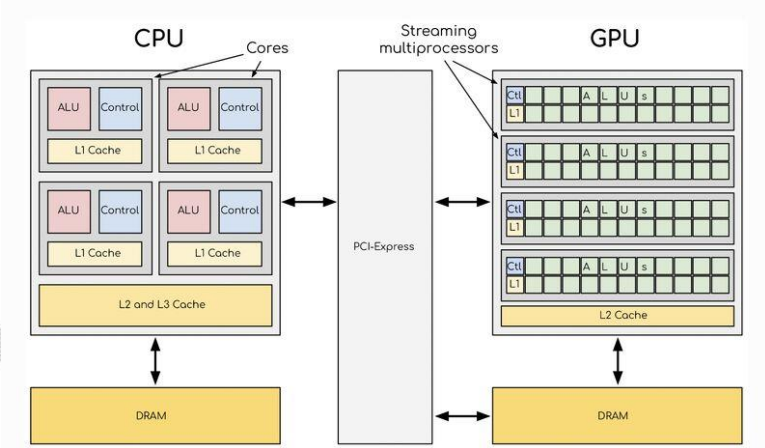
Purpose: Bridge speed gap between fast processors and slow memory

Hierarchy Levels:

- **Registers**: Fastest, smallest, processor-specific
- **L1, L2, L3 Cache**: On-chip, close to core, progressively larger and slower
- **Main Memory (RAM)**: Shared DRAM, slower than cache
- **Disk/SSD**: Persistent storage, very high latency

GPU-Specific:

- **Shared Memory**: Fast, limited, user-controlled
- **Global Memory**: Accessible by all threads, high latency
- **Constant/Texture Memory**: Specialized, cached memory for specific use cases



GPU Memory Hierarchy

GPU performance strongly depends on memory behavior.

Memory Types:

- **Registers (Fastest):** Private to each thread.
- **Shared Memory (Fast):**
  - Shared among threads within a block.
  - Low latency, programmer-controlled.
- **Global Memory (Slow):**
  - Accessible by all threads.
  - High latency (hundreds of cycles).

innovate achieve lead
Work Integrated Learning Programmes

GPU Memory Hierarchy

Performance Depends On:

Memory Coalescing

Adjacent threads should access adjacent memory locations:

Coalesced Access ⇒ Fewer memory transactions

Minimizing Global Memory Access

Use shared memory to reduce global access.

Maximizing Occupancy

Occupancy = Active threads / Maximum supported threads

Higher occupancy hides memory latency.

Work Integrated Learning Programmes

GPU Memory Hierarchy

GPU performance is dominated by:

Performance = Massive Thread Parallelism + High Memory Bandwidth

Unlike CPUs (latency-optimized), GPUs are:

Throughput-Optimized Processors

They perform best when:

- Large data size n
- High parallel fraction P
- Minimal branching
- Memory access is well-structured

Memory Bandwidth Considerations

Memory Bandwidth: Maximum rate at which data can be transferred between memory and processors

Why Important?

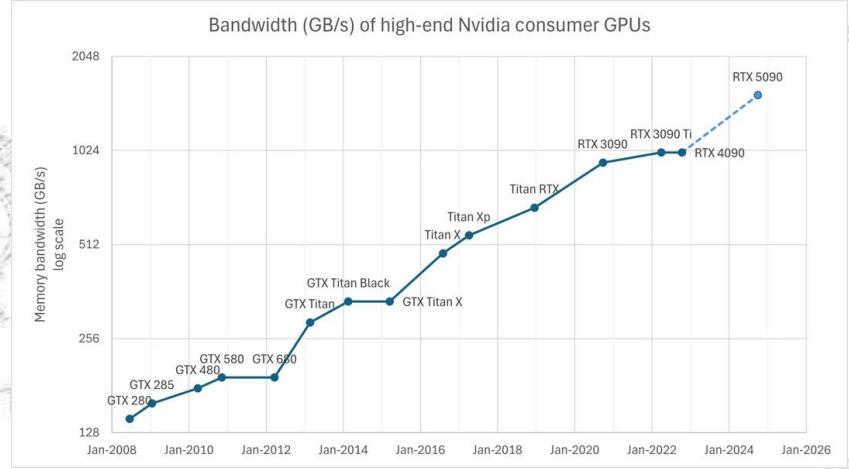
High computation speeds are meaningless without fast data access

Multicore CPUs:

- Shared bandwidth among cores → contention possible
- Cache reduces memory access latency

GPGPUs:

- Very high theoretical memory bandwidth
- Sensitive to **access patterns**
- **Coalesced access** (threads accessing adjacent memory) is critical for performance



Optimizing Memory Performance

For Multicore CPUs:

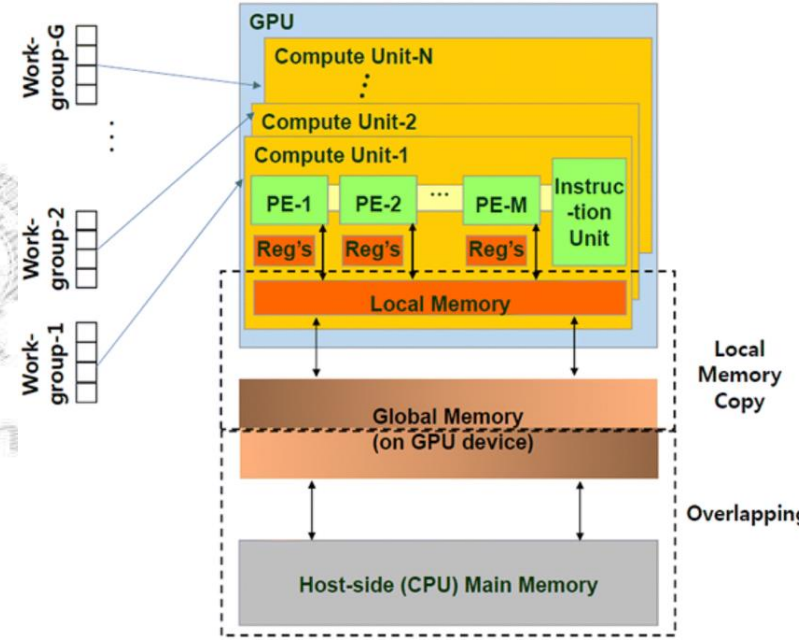
- Use **locality of reference** to benefit from caches
- Avoid **false sharing** – separate data written by different threads
- Use cache-aware data structures

For GPGPU:

- Align memory access for coalescing
- Utilize **shared memory** to avoid global memory latency
- Minimize **bank conflicts** in shared memory

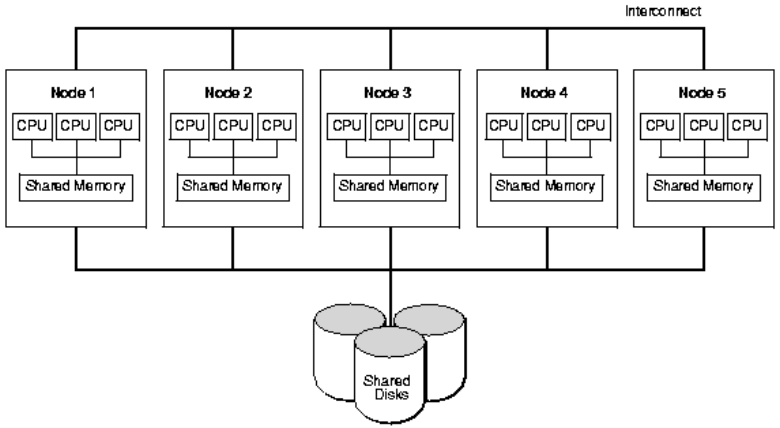
General Strategies:

- Balance **compute vs memory** loads
- Use **pipelining and tiling** techniques



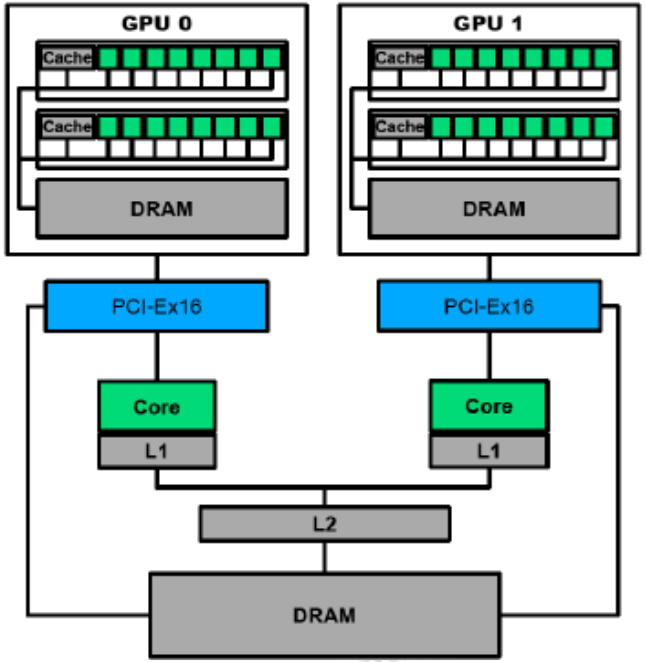
CPU-only Clusters

- Comprised solely of **multicore CPU-based nodes**.
- Suitable for:
  - Applications with control-heavy logic
  - Large in-memory databases
  - Distributed file systems and ETL pipelines
- **Benefits:**
  - More memory per node (better for large datasets)
  - Well-supported by traditional distributed frameworks (e.g., Hadoop, Spark, MPI)
  - Easier to manage and deploy
- **Limitations:**
  - Lower computational throughput
  - Slower performance in compute-intensive tasks like deep learning



GPU-Accelerated Clusters

- Consist of nodes with both CPUs and **one or more GPUs** (NVIDIA, AMD).
- Ideal for:
  - Training large-scale deep learning models (CNNs, Transformers)
  - Image/video processing, simulations
- **Benefits:**
  - Massive parallelism using thousands of CUDA cores
  - Faster matrix operations (e.g., GEMM, convolutions)
  - High floating-point throughput (TFLOPs range)
- **Limitations:**
  - Lower memory per GPU
  - Requires GPU-compatible libraries (e.g., TensorFlow GPU, CUDA, NCCL)
  - Higher hardware and maintenance cost



CPU-only vs GPU-Accelerated Clusters

Feature	CPU-only Clusters	GPU-Accelerated Clusters
Compute Units	Multicore CPUs	CPUs + Dedicated GPUs
Memory per Node	High	Lower (per GPU)
Programming Model	Threading, MPI	CUDA, OpenCL, TensorFlow, PyTorch
Power Consumption	Lower	Higher
Ideal Use Cases	Control-heavy tasks, I/O-bound jobs	Deep learning, simulations, HPC
Cost	Lower	Higher (GPUs, cooling systems, etc.)

Work Integrated Learning Programmes